The Journal of Systems and Software 83 (2010) 2275–2286

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A replicated and refined empirical study of the use of friends in C++ software

Michael English [a,*], Jim Buckley [a], Tony Cahill [b]

[a] Lero, CSIS Department, University of Limerick, Ireland
[b] CSIS Department, University of Limerick, Ireland

ARTICLE INFO

Article history:
Received 20 January 2010
Received in revised form 25 June 2010
Accepted 3 July 2010
Available online 15 July 2010

C++
Object-oriented
Empirical study
Friend
Open-source

ABSTRACT

The friend mechanism is widely used in C++ software even though the potential benefits of its use are disputed and little is known about when, where and why it is employed in practice. Furthermore, there is limited empirical analysis of its impact in object-oriented software, with only one study (Counsell and Newson, 2000) reported at journal level.

This paper aims to add to the empirical evidence of friendship's impact by replicating Counsell and Newson (2000)'s original study. The study's design is refined to improve the construct validity of the evaluation and a larger cohort of systems is used to improve the generalisability of the results. The findings suggest that classes involved in friendship are coupling hotspots and that there is no link between inheritance and friendship, contrary to the findings presented in Counsell and Newson (2000). The findings also suggest that the use of friends in a class is independent of the number of hidden members in a class.

1. Introduction

The friend construct is a C++ mechanism which grants a class or function access to the internal parts of other classes. Ellis and Stroustrup (1990) define a friend of a class as a method "that is not a member of the class but is permitted to use the private and protected member names from the class". Thus, the use of the friend construct results in the creation of coupling between classes.

The friend mechanism facilitates greater accessibility between classes in the C++ language. It has been suggested in the literature that the friend mechanism is the prevailing coupling mechanism used in software developed in the C++ language (Counsell et al., 2004). Additionally, since the use of the friend construct does not restrict coupling to occur through a class's interface, it is often considered poor programming practice. Coplien (1992) claims that the encapsulation provided by the protected and private mechanisms is violated by friendship. Page-Jones (1995) describes the use of friend methods to access the variables of a class directly as a "design sin", something which is usually done to improve efficiency. Similar perspectives can be found elsewhere (Deitel and Deitel, 2003; Lorenz and Kidd, 1994; Harrison et al., 1997). Indeed the use of the friend mechanism has been associated with defects in software, Briand et al. (1997b); Subramanyam and Krishnan (2003).

In contrast Cline (1991) states that if friends are used correctly then they enhance encapsulation. He suggests that sometimes a class needs to be split in half "when the two halves will have different numbers of instances or different lifetimes". In these circumstances each class typically needs direct internal access to each other and this is facilitated through the friend mechanism. Stroustrup (1994) also defends the friendship mechanism, describing it as "one protection domain granting a read-write capability to another". He argues that the declaration of a friend is part of the class declaration and rejects the argument that friendship violates encapsulation as "a combination of ignorance and confusion with non-C++ terminology". Meyers (1998) also holds the view that friends can be considered part of the class's interface, but recommends that friends should be avoided whenever possible. His advice is that if a method can be implemented in terms of the class's public interface, then it should not be made a friend. Booch (1991) in commenting on object modeling, suggests that a module's interface should be as narrow as possible and that friends should be chosen wisely, since friendship implies a certain trust between classes. Hence, the general perspective is a confused picture varying between unacceptability and acceptability when used appropriately (preferably by expert software designers).

Given this confusion, empirical investigation is imperative, especially considering the extensive use of the friend construct in software development in C++ (Counsell and Newson, 2000; Counsell et al., 2004; English et al., 2005b). In fact, Counsell et al. (2004) stated that library-based systems "showed a distinct lack of any form of coupling (including inheritance)" between classes "other than through the C++ friend facility". Application-based software also uses the friend construct regularly, although to a lesser extent than library-based systems (English et al., 2005b).

* Corresponding author. Tel.: +353 61 202772; fax: +353 61 202734.
E-mail address: michael.english@ul.ie (M. English).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.07.013

Counsell and Newson (2000) have empirically studied the use of the friend construct in four software systems and examined its relationship with inheritance and coupling. Given the prevalence of the friend mechanism in software, this study deserves revisiting. Counsell and Newson (2000) suggested that the friend construct has been used as a way of avoiding the use of inheritance, one of the distinguishing structural (and coupling) features of the OO paradigm. As such, there has been much discussion and disagreement as to the appropriate uses of inheritance, e.g. Singh (1995); Page-Jones (1995); Taivalsaari (1996); Willis (1996); Meyer (1997) with many empirical studies undertaken to examine the use of inheritance in software systems to address these disagreements, e.g. Bieman and Zhao (1995); Cartwright (1998); Chidamber et al. (1998); Harrison et al. (2000).

However, the extensive empirical analysis of inheritance contrasts markedly with the limited assessment of the use of the friend construct, even though this construct seems to be widely used. Therefore, there is a need for a more extensive investigation into the use of the friend construct in software systems developed in the C++ language.

Thus the reasons for focusing on empirical evaluation of the friend construct can be summarized as follows:

- the extensive use of friends in C++ software (Counsell et al., 2004; English et al., 2003).
- the debate surrounding the use of the friend mechanism and its interaction with the inheritance mechanism.
- the limited empirical evaluation of the use of the friend mechanism apart from Counsell and Newson (2000).
- to increase the construct validity of the seminal work in this area (Counsell and Newson, 2000) by implementing a number of refinements to the hypotheses.
- to enhance the external validity of the results. The study of Counsell and Newson (2000) examined four systems whereas the study reported in this paper considers a large cohort of both library- and application-based systems from a variety of domains.
- to increase the internal validity issue of causality by considering whether or not the size of classes in systems influences the results presented in the previous study as found in other empirical studies by other researchers (El Emam et al., 2001).

The structure of this paper is as follows. Section 2 reports on some related work which utilizes software metrics to investigate the internal characteristics of software systems. The study of the use of friends by Counsell and Newson (2000) is discussed in Section 3. The replication of the Counsell and Newson (2000) study is discussed in Section 3, including a description of the refinements implemented, the study design and the results. Section 5 concludes the paper with a summary of the findings and some avenues for future work.

2. Related work: empirical studies of software systems

Previously reported empirical studies of the internal attributes of software systems, including inheritance and the friend construct are discussed in Section 2.1. The importance of replication of empirical studies is highlighted below in Section 2.2.

2.1. Studying the internal attributes of software systems

Software metrics extracted from the source code of software can help to characterise that software. For example, metrics have helped to identify the characteristics of the friend construct, inheritance and coupling in software systems. For coupling in particular, many metrics have been presented in the literature (Chidamber and Kemerer, 1994; Harrison et al., 1998b; Henry and Kafura, 1981; Li and Henry, 1993; Wilkie and Hylands, 1998; Yu et al., 2002; Briand et al., 1997b). In addition, coupling measurement frameworks have been presented which facilitate measuring coupling at different levels of granularity (Wilkie and Kitchenham, 2000; Briand et al., 1997a).

In all coupling relationships between classes one class acts as a server and the other as a client. As a result coupling is measured from the perspective of the server (which exports services) and the client (which imports services). Many coupling metrics which have been defined take the direction of coupling into account (Henry and Kafura, 1981; Li and Henry, 1993; Wilkie and Hylands, 1998; Yu et al., 2002; Briand et al., 1997b). In fact coupling measures that amalgamate both directions of coupling are poor indicators (Yu et al., 2002). The extraction of these metrics from systems provides the data for the subsequent evaluation of hypotheses using statistical analysis techniques.

Several studies involving software metrics include an analysis of any relationships between the metrics themselves, usually via a correlation study. This analysis is normally undertaken in the context of a study to build a prediction model for some external attribute using measures of the internal characteristics (i.e. the measured metrics) of a system e.g. (Gyimothy et al., 2005; Briand et al., 2000).

Some studies examine sets of metrics in a bit more detail. Abreu and Capapuça (1994) studied the MOOD metrics suite applied to five software projects. They examined some summary statistical information (90% confidence interval) related to the metrics (Abreu et al., 1995) and suggested upper and lower bounds for these metrics. Succi et al. (2005) extracted and analysed the Chidamber and Kemerer (1994) metrics from 200 public domain projects. The authors recommended that metrics which showed high correlations with each other and those with low variance should not be included in a predictive model.

The use of inheritance in object-oriented systems has also been studied (Bieman and Zhao, 1995; Cartwright, 1998). They both conclude that inheritance is used to a limited extent, suggesting that alternative coupling mechanisms must be in widespread use in software systems.

Counsell and colleagues have performed a number of studies evaluating the structure and design of object-oriented systems and in particular their use of object-oriented facilities such as inheritance, encapsulation and the friend mechanism of C++ (Counsell and Newson, 2000; Counsell et al., 2000, 2002, 2004). In these studies, metrics are employed to facilitate the analysis and interpretation of a number of hypotheses.

One such study considered the relationship between inheritance and encapsulation by comparing the use of the protected and private language facilities, in classes which are engaged in inheritance and which are not engaged in inheritance (Counsell et al., 2002). Results indicated that stand-alone classes make considerable use of the protected mechanism, even though this feature is only useful for classes engaged in inheritance. Two possible reasons for this are suggested. The first is that these classes were removed from their inheritance hierarchies without the protected members being redefined. This may occur if the classes in question were key classes in the system and were removed from the hierarchy to allow easier access to their members (Counsell et al., 2002). The second possible reason for protected methods to appear in stand-alone classes is that it may have been anticipated that classes were designed with the intention of being part of an inheritance hierarchy at some future point.

Other studies by Counsell et al. (2000, 2004) compared the use of the coupling mechanisms (aggregation, association and generalisation) with the number of methods and attributes in a class. However, one common finding among these studies was the exten-